17

"Remaining calm solves great problems."

Processor

"Processor" and CPU (Central Processing Unit) refers the same—the heart of the computer. It is a chip that is responsible for processing instructions.

17.1 Processors

The computing world came across so many processors. Each of the processors has its own merits and demerits. The following table shows few of the known processors and its characteristics.

Date Introduced	Processor	Coprocessor	Internal Register size (bit)	Data I/O Bus width (bit)	Memory Address Bus width (bit)	Maximum Memory
June, 1978	8086	8087	16	16	20	1MB
June, 1979	8088	8087	16	8	20	1MB
Feb, 1982	286(80286)	80287	16	16	24	16MB
June, 1988	386 SX	80387 SX	32	16	24	16MB
April, 1989	486 DX	Built-in	32	32	32	4MB
March, 1993	Pentium	Built-in	32	64	32	4MB
May, 1997	Pentium II	Built-in	32	64	36	64MB

17.2 Processor Modes

When we look into the history of processors, two processors marked remarkable changes in computing, namely 8088 and 286. These processors are actually responsible for the so called 'processor modes'.

17.2.1 Real Mode

8088 processor is sometimes referred as 16-bit, because it could execute only 16-bit and could address only 1MB of memory instruction set using 16-bit registers. The processor introduced after 8088, namely 286 was also 16-bit, but it was faster than 8088. So these processors (8088 and 286) can handle only 16-bit software and operating systems like Turbo C++3.0, Windows 3.X, etc.

These processors had some drawbacks:

1. Normally didn't support multitasking
2. Had no protection for memory overwriting. So, there is even a chance to erase the operating system present in memory. In other words, 'memory crash' is unavoidable.

This 16bit instruction mode of 8088 and 286 processors are commonly known as 'Real Mode'.

Note
TC++3.0 is 16-bit. Therefore it is not preferred for commercial applications.

17.2.2 Protected Mode

The first 32-bit processor namely 386, has a built-in mechanism to avoid 'memory crash'. So this 32-bit mode is commonly known as *'protected mode'*. It also supports multitasking. UNIX, OS/2 and Windows NT are the pure 32-bit operating systems. 386 processor are also backward compatible, which means it could even handle 16-bit instructions and could even run on real mode.

17.2.3 Virtual Real Mode

When 386 processor was introduced, programmers were still using 16-bit instructions (real mode) on 386 because 386 executes the 16-bit application much faster. They also resisted 32-bit operating system and 32-bit applications. So when Microsoft tried to introduce Windows 95, a 32-bit operating system, it added a backward compatibility and introduced a mode called 'Virtual real mode'. That is, the programmer may think that it is working under real mode, but it is actually protected from hazardous effects.

17.3 Processor Type

Each processor has its own unique charactersitcs. When we check for its unique characteristics, we can find whether our processor is 286 or 386 or 586(Pentium). This logic is used to find out the processor type. Processor type is also referred as *CPU Id*.

17.3.1 C program to find processor type

Finding out the processor type using C program is difficult. Any how **Gilles Kohl** came out with a tough C code that can determine processor type (386 or 486).

```
int Test386( void )
{
   char far *p = "\270\001pP\235\234X\313";
```

```
   return!!(((int(far*)())p)
         ()&(( 0x88 + ((  286 | 386  )*4))<<4));
} /*--Test386( )--------*/

int main( void )
{
   printf( "Running on a %s\n", Test386() ? "386" : "286" );
   return(0);
} /*--main( )-----*/
```

If the code is run on a machine that don't have 386 or 486, you may get a wrong output. For better results we must use Assembly. (We can call it as a limitation of C language!).

17.3.2 Assembly routine to find processor type

The following Assembly routine is by **Alexander Russell**. Using this routine, we can find out our processor type and coprocessor support. This routine can be called from C i.e. you can link the object code with C program.

17.3.2.1 Assembly routines

To understand this Assembly module, read the comments provided in comment line.

```
;-------------------------------------------------------
; Hardware detection module
;
; Compile with Tasm.
; C callable.
;-------------------------------------------------------

.model medium, c

       global x_processor            :proc
       global x_coprocessor          :proc

LOCALS
.386


CPUID   MACRO
      db    0fh,     0A2h
ENDM

      .code

i86       equ 0
i186      equ 1
i286      equ 2
```

```
i386      equ 3
i486      equ 4
i586      equ 5

;---------------------------------------------
; PC Processor detection routine
;
; C callable as:
;    unsigned int x_processor( );
;
;
x_processor PROC
.8086
      pushf                    ; Save flags

      xor  ax,ax              ; Clear AX
      push ax                  ; Push it on the stack
      popf                     ; Zero the flags
      pushf                    ; Try to zero bits 12-15
      pop  ax                  ; Recover flags
      and  ax,0F000h           ; If bits 12-15 are 1 => i86 or i286
      cmp  ax,0F000h
      jnz  @@not_86_186
      jmp  @@is_86_186

@@not_86_186:

      mov  ax,07000h           ; Try to set bits 12-14
      push ax
      popf
      pushf
      pop  ax
      and  ax,07000h           ; If bits 12-14 are 0 => i286
      jnz   is_not_286
      jmp is_286

is_not_286:


      ; its a 386 or higher

      ; check for 386 by attempting to toggle EFLAGS register
      ; Alignment check bit which can't be changed on a 386
.386
      cli
      pushfd
      pushfd
```

```
        pop   eax
        mov  ebx, eax
        xor   eax, 040000h      ; toggle bit 18
        push  eax
        popfd
        pushfd
        pop   eax
        popfd
        sti
        and   eax, 040000h      ; clear all but bit 18
        and   ebx, 040000h      ; same thing
        cmp   eax, ebx
        jne   @@moretest
        mov   ax, i386
        jmp short @@done

        ; is it a 486 or 586 or higher

@@moretest:

        ; check for a 486 by trying to toggle the EFLAGS ID bit
        ; this isn't a foolproof check

        cli
        pushfd
        pushfd
        pop   eax
        mov   ebx, eax
        xor   eax, 0200000h      ; toggle bit 21
        push  eax
        popfd
        pushfd
        pop   eax
        popfd
        sti
        and   eax, 0200000h      ; clear all but bit 21
        and   ebx, 0200000h      ; same thing
        cmp   eax, ebx
        jne   @@moretest2
        mov   ax, i486
        jmp short @@done

@@moretest2:

        ; OK it was probably a 486, but let’s double check

        mov   eax, 1
```

```
        CPUID
        and   eax, 0f00h
        shr   eax, 8

        mov   ebx, eax
        mov   ax, i586
        cmp   ebx, 5
        je    @@done    ; it was a pentium

        ; it wasn't a 586 so just report the ID

        mov   eax, ebx
        and   eax, 0ffffh

        jmp  short @@done

.8086

is_286:
        mov  ax,i286              ; We have a 286
        jmp  short @@done

@@is_86_186:                      ; Determine whether i86 or i186
        push cx                   ; save CX
        mov  ax,0FFFFh            ; Set all AX bits
        mov  cl,33                ; Will shift once on 80186
        shl  ax,cl                ; or 33 x on 8086
        pop  cx
        jnz  is_186               ; 0 => 8086/8088
is_86:
        mov  ax,i86
        jmp  short @@done
is_186:
        mov  ax,i186
@@done:
        popf

        ret

x_processor endp

.386

 .8086
;---------------------------------------------
; PC Numeric coprocessor detection routine
;
```

```
; C callable as:
;    unsigned int x_coprocessor( );
;
;  Returns 1 if coprocessor found, zero otherwise

x_coprocessor PROC

      LOCAL     control:word

      fninit                           ; try to initialize the copro.
      mov    [control],0               ; clear control word variable
      fnstcw control                   ; put control word in memory
      mov    ax,[control]              ;
      cmp    ah,03h                    ; do we have a coprocessor ?
      je     @@HaveCopro               ; jump if yes!
      xor    ax,ax                     ; return 0 since nothing found
      jmp    short @@Done
@@HaveCopro:
      mov    ax,1
@@Done:
      ret

x_coprocessor   endp


end
;---------------------------
```

17.3.2.2 Calling C program

```
#pragma –mm       /* force to medium memory model */

int main( void )
{
   int i;
   static char *cpu_str[]=
            {
               "i86",
               "i186",
               "i286",
               "i386",
               "i486",
               "i586",
               "i686"
            };

   i = x_processor( );
```

```
   if ( i > 6 )
            i = 6;

   printf( "Processor type: %s   CoPro : %s\n", cpu_str[i],
            x_coprocessor( ) ? "Yes" : "No");
   return(0);
} /*--main( )----------*/
```

17.3.3 Another Assembly routine

The success of the above Assembly code by Alexander Russell depends on the code that the compiler produces. So if your compiler doesn't produce the "right" code, you may not get proper results. Here I provide another Assembly code to find out processor type. It is by **Edward J. Beroset**. All these codes use the same logic i.e. checking the unique characteristics of a processor.

This module contains a C callable routine which returns a 16-bit integer (in AX) which indicates the type of CPU on which the program is running. The lower eight bits (AL) contain a number corresponding to the family number (e.g. 0 = 8086, 1 = 80186, 2 = 80286, etc.). The higher eight bits (AH) contain a collection of bit flags which are defined below.

```
; cpuid.asm
;
%       .MODEL  memodel,C               ;Add model support via command
                                        ;line macros, e.g.
                                        ;MASM /Dmemodel=LARGE,
                                        ;TASM /Dmemodel=SMALL, etc.
        .8086
        PUBLIC cpu_id

;
; using MASM 6.11       Ml /c /Fl CPUID.ASM
;
; using TASM 4.00       TASM CPUID.ASM
;
; using older assemblers, you may have to use the following equate
; and eliminate the .586 directive
;
;CPUID equ "dw 0a20fh"
;
; bit flags for high eight bits of return value
;
HAS_NPU         equ     01h
IS386_287       equ     02h
IS386SX         equ     04h
CYRIX           equ     08h
```

```
NEC             equ     10h
NEXGEN          equ     20h
AMD             equ     40h
UMC             equ     80h

        .code

cpu_id  proc
        push    bx
        push    cx
        push    dx
        push    bp
        mov     bp,sp
        xor     dx,dx                   ; result = 0 (UNKNOWN)
;*********************************************************************
; The Cyrix test
;
;   Cyrix processors do not alter the AF (Aux carry) bit when
;   executing an XOR.  Intel CPUs (and, I think, all the others)
;   clear the AF flag while executing an XOR AL,AL.
;
;*********************************************************************
TestCyrix:
        mov     al,0fh                  ;
        aas                             ; set AF flag
        xor     al,al                   ; only Cyrix leaves AF set
        aas                             ;
        jnc     Test8086                ;
        or      dh,CYRIX                ; it's at least an 80386 clone
        jmp     Test486                 ;
;*********************************************************************
;
; The 80186 or under test
;
;   On <80286 CPUs, the SP register was decremented *before* being
;   pushed onto the stack.  All later CPUs do it correctly.
;
;*********************************************************************
Test8086:
        push    sp                      ; Q: is it an 8086, 80188, or
        pop     ax                      ;
        cmp     ax,bp                   ;
        je      Test286                 ;   N: it's at least a 286
;*********************************************************************
; The V20/V30 test
;
;   NEC's CPUs set the state of ZF (the Zero flag) correctly after
```

```
;   a MUL.  Intel's CPUs do not -- officially the state of ZF is
;   "undefined" after a MUL or IMUL.
;
;*********************************************************************
TestV20:
        xor     al,al                   ; clear the zero flag
        mov     al,1                    ;
        mul     al                      ;
        jnz     Test186                 ;
        or      dh,NEC                  ; it's a V20 or a V30
;*********************************************************************
; The 80186 test
;
;   On the 80186, shifts only use the five least significant bits,
;   while the 8086 uses all 8, so a request to shift 32 bits will
;   be requested as a shift of zero bits on the 80186.
;
;*********************************************************************
Test186:
        mov     al,01h                  ;
        mov     cl,32                   ; shift right by 33 bits
        shr     al,cl                   ;
        mov     dl,al                   ; al = 0 for 86, al = 1 for 186
longTestNpu:
        jmp     TestNpu                 ;

;*********************************************************************
; The 286 test
;   Bits 12-15 (the top four) of the flags register are all set to
;   0's on a 286 and can't be set to 1's.
;
;*********************************************************************
Test286:
        .286
        mov     dl,2                    ; it's at least a 286
        pushf                           ; save the flags
        pop     ax                      ; fetch 'em into AX
        or      ah,0f0h                 ; try setting those high bits
        push    ax                      ;
        popf                            ; run it through the flags reg
        pushf                           ;
        pop     ax                      ; now check the results
        and     ah,0F0h                 ; Q: are bits clear?
        jz      longTestNpu             ;   Y: it's a 286

;*********************************************************************
; The 386 test
```

```
;
;   The AC (Alignment Check) bit was introduced on the 486.  This
;   bit can't be toggled on the 386.
;
;*********************************************************************
Test386:
        .386
        mov     dl,3                    ; it's at least a 386
        pushfd                          ; assure enough stack space
        cli
        and     sp, NOT 3               ; align stack to avoid AC fault
        pushfd                          ;
        pop     cx                      ;
        pop     ax                      ;
        mov     bx,ax                   ; save a copy
        xor     al,4                    ; flip AC bit
        push    ax                      ;
        push    cx                      ;
        popfd                           ;
        pushfd                          ;
        pop     cx                      ;
        pop     ax                      ;
        and     al,4                    ;
        sti
        xor     al,bl                   ; Q: did AC bit change?
        jnz     Test486                 ;   N: it's a 386
        .386P
;*********************************************************************
; The 386SX test
;
;   On the 386SX, the ET (Extension Type) bit of CR0 is permanently
;   set to 1 and can't be toggled.  On the 386DX this bit can be
;   cleared.
;*********************************************************************
        mov     eax,cr0
        mov     bl,al                   ; save correct value
        and     al,not 10h              ; try clearing ET bit
        mov     cr0,eax                 ;
        mov     eax,cr0                 ; read back ET bit
        xchg    bl,al                   ; patch in the correct value
        mov     cr0,eax                 ;
        test    bl,10h                  ; Q: was bit cleared?
        jz      TestNpu                 ;  Y: it's a DX
        or      dh,IS386SX              ;  N: it's probably an SX

;*********************************************************************
```

```
; The 486 test
;
;   Try toggling the ID bit in EFLAGS.  If the flag can't be toggled,
;   it's a 486.
;
; Note:
;   This one isn't completely reliable -- I've heard that the NexGen
;   CPU's don't make it through this one even though they have all
;   the Pentium instructions.
;***********************************************************************
Test486:
        .486
        pushfd
        pop     cx
        pop     bx
        mov     dl,4                    ;
        mov     ax,bx                   ;
        xor     al,20h                  ; flip EFLAGS ID bit
        push    ax                      ;
        push    cx                      ;
        popfd                           ;
        pushfd                          ;
        pop     cx                      ;
        pop     ax                      ;
        and     al,20h                  ; check ID bit
        xor     al,bl                   ; Q: did ID bit change?
        jz      TestNpu                 ;   N: it's a 486

;***********************************************************************
; The Pentium+ tests
;
;   First, we issue a CPUID instruction with EAX=0 to get back the
;   manufacturer's name string.  (We only check the first letter.)
;
;***********************************************************************
PentPlus:
        .586
        push    dx                      ;
        xor     eax,eax                 ;
        cpuid                           ;
        pop     dx                      ;
        cmp     bl,'G'                  ; Q: GenuineIntel?
        jz      WhatPent                ;   Y: what kind?
        or      dh,CYRIX                ; assume Cyrix for now
        cmp     bl,'C'                  ;
        jz      WhatPent                ;
        xor     dh,(CYRIX OR AMD)       ;
```

```
        cmp     bl,'A'                  ;
        jz      WhatPent                ;
        xor     dh,(AMD OR NEXGEN)      ;
        cmp     bl,'N'                  ;
        jz      WhatPent                ;
        xor     dh,(NEXGEN OR UMC)      ; assume it's UMC
        cmp     bl,'U'                  ;
        jz      WhatPent                ;
        xor     dh,UMC                  ; we don't know who made it!
;**********************************************************************
; The Pentium+ tests (part II)
;
;   This test simply gets the family information via the CPUID
;   instruction
;
;**********************************************************************
WhatPent:
        push    edx                     ;
        xor     eax,eax                 ;
        inc     al                      ;
        cpuid                           ;
        pop     edx                     ;
        and     ah,0fh                  ;
        mov     dl,ah                   ; put family code in DL

;**********************************************************************
; The NPU test
;
;   We reset the NPU (using the non-wait versions of the instruction,of
;   course!), put a non-zero value on the stack, then write the NPU
;   status word to that stack location.  Then we check for zero, which
;   is what would be there if there were an NPU.
;
;**********************************************************************
TestNpu:
        .8087
        .8086
        mov     sp,bp                   ; restore stack
        fninit                          ; init but don't wait
        mov     ax,0EdEdh               ;
        push    ax                      ; put non-zero value on stack
        fnstsw  word ptr [bp-2]         ; save NPU status word
        pop     ax                      ;
        or      ax,ax                   ; Q: was status = 0?
        jnz     finish                  ;   N: no NPu
        or      dh,HAS_NPU              ;   Y: has NPU
```

```
;*********************************************************************
; The 386/287 combo test
;
;   Since the 386 can be paired with either a 387 or 287, we check to
;   see if the NPU believes that +infinity equals -infinity.  The 387
;   says they're equal, while the 287 doesn't.
;
;*********************************************************************
        cmp     dl,3                    ; Q: is CPU a 386?
        jnz     finish                  ;  N: no need to check
infinities
        fld1                            ; load 1
        fldz                            ; load 0
        fdiv                            ; calculate infinity! (1/0)
        fld     st                      ; duplicate it
        fchs                            ; change signs of top inf
        fcompp                          ; identical?
        push    ax                      ;
        fstsw   word ptr [bp-2]         ;
        pop     ax                      ;
        test    ah,40h                  ; Q: does NPU say they're
equal?
        jz      finish                  ;  N: it's a 387
        or      dh,IS386_287            ;
finish:
        mov     ax,dx                   ; put our return value in place
        pop     bp                      ; clean up stack
        pop     dx                      ;
        pop     cx                      ;
        pop     bx                      ;

        ret                             ;
cpu_id  endp

        END
;---------------------------
```

Exercises

1. Write a program that can find the current mode of processor (i.e., Real / Protected / Virtual Mode).